Otimização de Redes Neurais usando Algoritmos Genéticos

Guilherme Augusto Anício Drummond do Nascimento[1], **Rafael Augusto Freitas Oliveira**[1]

[1]Departamento de Computação – Universidade Federal de Ouro Preto (UFOP)
Ouro Preto – MG – Brasil

Abstract. *This article addresses the optimization of hyperparameters in Convolutional Neural Networks (CNNs) using Genetic Algorithms (GAs). Hyperparameters play a crucial role in the effectiveness of machine learning, and their manual optimization can be tedious. The focus is to optimize a CNN for image recognition on the MNIST and Fashion-MNIST datasets. The literature review explores fundamental concepts such as CNNs and GAs, along with related works. The proposed methodology involves optimizing various hyperparameters, such as the number of filters, kernel sizes, and activation functions. The goal is to maximize CNN performance through efficient hyperparameter optimization.*
Keywords*: Hyperparameter Optimization, Convolutional Neural Networks, Genetic Algorithms, Machine Learning.*

Resumo. *O artigo aborda a otimização de hiperparâmetros em Redes Neurais Convolucionais (CNNs) usando Algoritmos Genéticos (AGs). Hiperparâmetros desempenham um papel crucial na eficácia do aprendizado de máquina, e sua otimização manual pode ser tediosa. O foco é otimizar uma CNN para reconhecimento de imagens dos conjuntos MNIST e Fashion-MNIST. A revisão bibliográfica explora conceitos fundamentais, como CNNs e AGs, além de trabalhos relacionados. A metodologia proposta envolve a otimização de diversos hiperparâmetros, como número de filtros, tamanhos de kernels e funções de ativação. O objetivo é maximizar a performance da CNN por meio da otimização eficiente dos hiperparâmetros.*
Palavras-chave*: Otimização de Hiperparâmetros, Redes Neurais Convolucionais, Algoritmos Genéticos, Aprendizado de Máquina.*

1. Introdução

Parâmetros são otimizados durante certas etapas da modelagem, como os pesos dos neurônios durante a fase de treinamento. Porém, os hiperparâmetros são escolhidos antes do treinamento, durante a construção da estrutura e do comportamento do modelo. Por exemplo, número de camadas, taxa de aprendizado, número de épocas, tamanho dos lotes, entre outros. Em outras palavras, um hiperparâmetro é um parâmetro cujo valor é usado para controlar o processo de aprendizagem. Em contrates, os valores de outros parâmetros, como peso dos neurônios são derivados pelo treinamento.

Hiperparâmetros não podem ser inferidos durante o processo de adequar o modelo ao conjunto de treinamento, pois se referem a tarefa de seleção de parâmetros ou hiperparâmetros de algoritmos, que afetam a velocidade e qualidade do processo de treinamento [Faceli et al. 2021]. Otimizar hiperparâmetros, então, é o problema de escolher um conjunto de hiperparâmetros ótimos para um algoritmo de aprendizagem.

A otimização desses hiperparâmetros é um processo tedioso, em que especialistas precisam configurar um conjunto de escolhas de hiperparâmetros manualmente. No entanto, diferentes conjuntos de dados requerem diferentes modelos ou combinação de hiperparâmetros, o que pode ser complicado e tedioso.

1.1. Objetivo

O objetivo deste artigo é apresentar uma proposta de otimização de hiperparâmetros utilizando algoritmos genéticos. Nosso foco será em otimizar uma Rede Neural Convolucional (*Convolutional Neural Networks* - CNN) para o reconhecimento de imagens dos datasets MNIST e Fashion-MNIST.

2. Revisão bibliográfica

Este capitulo apresenta uma base teórica para nossa proposta na Seção 2.1 e uma análise de trabalhos relacionados na Seção 2.2.

2.1. Fundamentação teórica

As Redes Neurais Convolucionais são uma das abordagens mais populares de redes profundas (DNs - *Deep Networks*) pré-treinadas [Faceli et al. 2021]. Elas imitam o processamento de imagem realizado pelo cérebro, onde de pequenos detalhes, como linhas e curvas, são extraídos padrões de crescente complexidade. As CNNs são compostas por dois estágios de processamento. No primeiro ela possui uma sequencia de pares de camadas: convolucional, que extrai mapas de características utilizando filtros, e uma camada de *pool*, que mantém apenas as informações mais relevantes. Já o segundo estágio é geralmente composto por uma rede Perceptron Multicamadas (MLP — *Multi Layer Perceptron*), possibilitando que as características extraídas sejam utilizadas como atributos preditivos para a rede MLP.

Os Algoritmos Genéticos (AGs) são uma categoria de técnicas dentro da Computação Evolutiva (CE) utilizados para resolver problemas baseados na Genética e na Teoria da Evolução [Faceli et al. 2021]. Esses algoritmos buscam encontrar soluções otimizadas para um determinado problema por meio da simulação de processos evolutivos. No contexto dos AGs, uma solução é representada como um indivíduo ou cromossomo, geralmente utilizando vetores binários, onde cada elemento (gene) indica a presença ou ausência de uma característica. O processo de busca por uma solução ocorre ao longo de várias gerações, utilizando operadores genéticos como seleção, cruzamento e mutação. Na seleção, os indivíduos mais aptos são probabilisticamente escolhidos para a reprodução, onde o cruzamento combina partes de pares de soluções e a mutação altera aleatoriamente componentes das soluções. O operador de elitismo preserva o melhor indivíduo automaticamente. Essa abordagem é aplicada iterativamente até que um critério de parada seja alcançado, proporcionando uma busca eficiente e simultânea em diversas regiões do espaço de soluções, tornando os AGs métodos de otimização global, embora seu custo computacional seja uma desvantagem, que pode ser mitigada pela paralelização.

2.2. Trabalhos Relacionados

[Yu and Zhu 2020] faz uma comparação de diversos algoritmos e aplicações para otimização de hiperparâmetros (OHP), analisando sua acurácia, eficiência e escopo da

aplicação, e categoriza os hiperparâmetros em relacionados à estrutura e relacionados ao treinamento, discutindo sua importância e estratégias empíricas para determinar quais hiperparâmetros estarão envolvidos no processo de OHP. Dentre os hiperparâmetros relacionados ao treinamento, os mais relevantes são a taxa de aprendizada (*learning rate* - LR) e aqueles ligados ao algoritmo de otimização, como a escolha do algoritmo, tamanho do *mini-batch* e *momentum*. Atualmente, o algoritmo de otimização, ou otimizador, mais usado é a descida de gradiente estocástica com *momentum* e suas variações como Ada-Grad, RMSprop e Adam. Além disso, uma alternativa a LR fixada é variar a LR durante o treinamento, chamado de *LR schedule* ou *LR decay*. Com esse método, a taxa de aprendizado diminui ao longo do processo de treinamento, o que envolve a escolha de outro hiperparâmetro: referente a quão rápido a LR vai diminuir. Os hiperparâmetros relacionados à arquitetura do modelo geralmente influenciam na capacidade de aprendizado e os exemplos mais comum são o número de camadas ocultas e a largura das camadas. O número de camadas ocultas é um parâmetro crítico para determinar a estrutura geral das redes neurais, que tem uma influência direta no resultado final, e o número de neurônios em cada camada também deve ser considerado de forma cuidados, pois a falta de neurônios podem causar *underfitting*, já que o modelo não é suficientemente complexo, enquanto neurônios demais podem tornar o modelo complexo demais, levando ao *overfitting*. Junto disso, é possível aplicar camadas de regularização, como a regularização *L1* e a *L2*, *data augmentation* ou *dropout*, para reduzir a complexidade dos modelos, especialmente daqueles com dados de treinamentos insuficientes.

O artigo de [Buarque et al. 2022] aplica Algoritmo Genético (GA) e *Particle Swarm Optimization* (PSO) para encontrar hiperparâmetros da rede neural Perceptron Multicamadas (MLP) com o objetivo de classificar Requisitos Não Funcionais. Utilizando Python e Keras, o artigo aplica os algoritmos a base PROMISE_exp que contem 969 requisitos de software, sendo 444 funcionais e 525 não-funcionais, divididos em 11 classes. O algoritmo genético foi implementado através da biblioteca DEAP, garantindo que os 3 melhores cromossomos estivessem na próxima geração sem sofrer alteração e após 30 execuções, o GA conseguiu gera 15.822 combinações com uma média de F1 de 0.6214, e o PSO conseguiu o maior resultado, com um F1 de 0.6426.

[Fernandes et al. 2022] apresentam um modelo de Redes Neurais Profundas (Deep Neural Networks - DNNs) para a detecção de cardiomegalia em radiografias de tórax que foi treinado usando otimização de hiperparâmetros. Cardiomegalia é um termo para representar uma variedade de condições causadoras do aumento do coração, que geralmente permanece sem diagnóstico até que apareçam os sintomas, tornando essencial a prevenção e tratamento. O trabalho apresenta uma metodologia DNNs em contra partida à literatura que comumente utiliza CNNs. Aplicada à *CheXpert Dataset*, uma base de dados criada em 2019 que contem imagens frontais e/ou laterais de Raio-X do tórax de pacientes. A otimização foi realizada com apoio da biblioteca *HyperOpt* baseado no *Sequential Model-Based Optimization* (SMBO), otimizando os hiperparâmetros *Dropout*, *Learn Rate*, *Batch Size*, *Optmizer* e *Epoch*, visando maximizar a métrica de especificidade e minimizar a *Loss* do modelo. As arquiteturas com melhores resultados foram as Xception e InceptionV3, que alcançaram um *Area Under the Curve* (AUC) de 0.919, uma acurácia de 0.873, um F1-Score de 0.876, uma sensibilidade de 0.913 e uma especificidade de 0.790.

[Aszemi and Dominic 2019] apresentam uma abordagem para redes neurais convolucionais (Convolutional Neural Networks - CNN) voltada para conjuntos de dados CIFAR-10[1], que hibridiza algoritmos genéticos com o método de procura local (*local search method*) na otimização das estruturas das redes e dos algoritmos de treinamento. Experimentos iniciais com CNNs pequenas na CPU produziram resultados com acurácia de apenas 60.85% e demoraram 5 dias para rodar, otimizando apenas a estrutura das redes, a taxa de aprendizado e tamanho dos *batchs*. Experimentos seguintes, ainda com CNNs pequenas, mas agora na GPU, e adicionando funções de ativação, otimizadores com *momentum* para normalização de *batch* e uma terceira camada de convolução. Tais experimentos rodaram em 3 dias e aumentaram a acurácia para 71.17%. No geral, a *random search* não alcançou acurácia do estado da arte, de 90% ou mais, mas alcançou resultados satisfatório, acima de 80%, na tabela de classificações do CIFAR-10.

O artigo de [Barbosa 2018] apresenta a aplicação de 3 versões de Algoritmos Genéticos (GAs) para otimizar a busca de hiperparâmetros em algoritmos de árvores de decisão. O conjunto de dados foi aplicado ao algoritmo *Classification and Regression Trees* (CART) para fins de classificação. Já na parte dos GAs, buscou-se otimizar a acurácia e o tamanho final da árvore gerada. Uma das variações utilizadas no artigo, é o *Fluid Genetic Algorithm* (FGA), proposto por [Jafari-Marandi and Smith 2017], que afirma que o FGA possui uma melhor taxa de sucesso, controle de convergência e versatilidade, pois é aplicável a problemas multiobjetivo e multinível, em comparação ao GA padrão. Ele possui duas principais diferença, a primeira é que cromossomos e indivíduos não são uma única entidade, que é mais próximo da realidade biológica, e a segunda diferença, é que no FGA não existe a mutação, pois o algoritmo forecene uma diversidade populacional inteligente. A ultima variação é a *Genetic Algorithm using Theory of Chaos* (GATC), que é baseado na teoria do caos de Edward Norton Lorenz, aplicado quando a ciência lida com problemas não-lineares, onde os comportamentos são impossíveis de prever ou controlar. Sua principal diferença está no processo de cruzamento, onde é aplicada a função caótica, e na representação cromossomial, onde o cromossomo é dividido em 3 partes, a solução do problema, o valor λ e a máscara de cruzamento uniforme.

3. Metodologia

Neste trabalho usaremos a biblioteca DEAP[2] do Python e uma implementação simples de um algoritmo genético para otimizar os hiperparâmetros de uma CNN para classificação dos conjuntos de dados MNIST ([LeCun et al. 2010]) e Fashion-MNIST ([Xiao et al. 2017]) e comparar os resultados obtidos pelas diferentes abordagens. Ambos os *datasets* estão disponíveis para uso no TensorFlow.

A rede a ser otimizada será composta por 8 camadas (sendo 6 ocultas), descritas a seguir:

1. A camada de entrada, que recebe $28 \times 28 \times 1$ pixels, o tamanho das imagens em escala de cinza dos datasets.
2. Uma camada de convolução.

[1]CIFAR-10 é um conjunto de dados que consiste de 60000 imagens RGB de 32x32 em 10 classes, com 6000 imagens por classe, coletados por Alex Krizhevsky, Vinod Nair e Geoffrey Hinton. Disponível em `https://www.cs.toronto.edu/~kriz/cifar.html`

[2]Disponível em `https://deap.readthedocs.io/en/master/`

3. Uma camada de *pooling*.
4. Uma camada de convolução.
5. Uma camada de *pooling*.
6. Uma camada de *flatten*.
7. Uma camada densa.
8. A camada de saída, com 10 neurônios, um para cada classe.

Os hiperparâmetros que serão otimizados para cada tipo de camada estão descritos a seguir:

- Camadas de convolução: número de filtros, tamanho do *kernel*, função de ativação.
- Camadas de *pooling*: tipo de *pooling*, tamanho do *kernel*, função de ativação.
- Camada densa: número de neurônios, função de ativação.

Na tabela a seguir são apresentados os diferentes valores para cada hiperparâmetro.

Camada	Hiperparâmetro	Valores
Camada 2 (convolução)	Número de filtros	[32, 64, 128, 256]
	Tamanho dos *kernels*	$[(3 \times 3), (5 \times 5), (7 \times 7)]$
	Função de ativação	[ReLU, GELU, *softmax*]
Camada 3 (*pooling*)	Tipo de *pooling*	[*max pooling*, *average pooling*]
	Tamanho do kernel	$[(2 \times 2), (3 \times 3)]$
Camada 4 (convolução)	Número de filtros	[8, 16, 32, 64]
	Tamanho dos *kernels*	$[(3 \times 3), (5 \times 5), (7 \times 7)]$
	Função de ativação	[ReLU, GELU, *softmax*]
Camada 5 (*pooling*)	Tipo de *pooling*	[*max pooling*, *average pooling*]
	Tamanho do kernel	$[(2 \times 2), (3 \times 3), (4 \times 4)]$
Camada 7 (densa)	Número de neurônios	[32, 64, 128, 256]
	Função de ativação	[ReLU, GELU, *softmax*, *sigmoid*]
Camada 8 (saída)	Função de ativação	[*softmax*, *sigmoid*]

Tabela 1. Tabela mostrando os valores testados para cada hiperparâmetro otimizado.

4. Desenvolvimento

Para o desenvolvimento do algoritmo, foi optado pelo uso da linguagem *python*, no ambiente de execução *colab*[3], e o uso da biblioteca DEAP[4](*Distributed evolutionary algorithms in python*). O objetivo é encontrar a combinação ideal de hiperparâmetros que resulte na melhor performance da CNN para classificação das bases de dados *MNIST* e *Fashion MNIST*.

Antes de iniciar o processo de otimização, foi necessário preparar as bases de dados *MNIST* e *Fashion MNIST*. Ambas são amplamente utilizadas em tarefas de reconhecimento de padrões e consistem em imagens de 28x28 pixels representando dígitos

[3]Disponível em: `https://colab.research.google.com/`
[4]Disponível em: `https://deap.readthedocs.io/en/master/`

escritos à mão (*MNIST*) e imagens de roupas (*Fashion MNIST*). Para esta etapa, utilizamos a biblioteca *TensorFlow* para carregar e normalizar os dados.

A arquitetura da CNN foi escolhida como ponto de partida para a otimização dos hiperparâmetros. Definimos uma arquitetura básica, composta por camadas convolucionais, de *pooling* e densas, e decidimos otimizar os hiperparâmetros apresentados na tabela anterior, porem por uma limitação do algoritmo, não foi possível otimizar o tamanho do *kernel* das camadas de *poolings*.

O algoritmo utilizado foi o *eaSimples*. Ele é uma implementação simples, mas poderosa, de um algoritmo genético dentro da biblioteca DEAP. Ele é frequentemente usado para otimização de problemas em que uma população de soluções candidatas evolui ao longo de várias gerações, com base em operadores genéticos como cruzamento e mutação. Durante cada geração, os indivíduos são avaliados de acordo com uma função de adaptação que quantifica sua qualidade em relação ao problema em questão. Os indivíduos mais aptos são selecionados para reprodução, gerando descendentes que herdam características benéficas dos pais. Com o tempo, o algoritmo converge para soluções que representam um ótimo ou subóptimo global, dependendo da natureza do problema e das restrições impostas. O *eaSimples* é fácil de usar e configurar, tornando-o uma escolha popular para uma variedade de aplicações de otimização.

Para fins de comparação, originalmente seriam comparados o algoritmo *eaSimples*, com os outros de mesma biblioteca: O *eaMuPlusLambda*, o *eaMuCommaLambda* e o *eaGenerateUpdate*, cada um com abordagens distintas para a atualização da população de soluções candidatas ao longo das gerações. O *eaMuPlusLambda* cria uma nova população combinando os indivíduos da população atual com os descendentes gerados, selecionando os melhores entre eles. Em contrapartida, o *eaMuCommaLambda* substitui completamente a população anterior pela nova população de descendentes, selecionando os melhores indivíduos entre eles. Já o *eaGenerateUpdate* atualiza a população diretamente a partir de uma lista de indivíduos gerados a cada geração, sem a necessidade de especificar um tamanho fixo para a população. Cada abordagem oferece vantagens e é mais adequada para diferentes cenários de problema e estratégias de otimização.

Porem, são algoritmos com um consumo muito alto de recursos computacionais, e devido a isso, foi necessário modificar nossas comparações. Escolhemos por comparar o *eaSimples* com um algoritmo de implementação própria. O código implementa um algoritmo genético para otimizar os hiperparâmetros de uma CNN. Ele começa criando uma população de indivíduos representando diferentes configurações de hiperparâmetros. Cada indivíduo é avaliado usando a acurácia como medida de aptidão, obtida ao treinar e validar o modelo da CNN com os hiperparâmetros do indivíduo. Os melhores indivíduos são selecionados com base em sua aptidão para reprodução, onde pares de indivíduos se cruzam para criar novos indivíduos com combinações de hiperparâmetros. Uma pequena porcentagem da população sofre mutação para introduzir diversidade. Esses passos são repetidos por várias gerações até que uma condição de parada seja alcançada, permitindo que o algoritmo explore e otimize eficientemente o espaço de busca de hiperparâmetros em busca de uma solução ótima ou próxima ótima.

5. Resultados

5.1. Algoritmos Genéticos

O algoritmo *eaSimples* recebe vários parâmetros de entrada, como foco para a *toolbox*(caixa de ferramentas), onde é armazenado todas as funções que serão utilizadas durante gerações, como fator de mutação, probabilidade de reprodução, entre outros. Colocamos como entrada, os dados que queremos que sejam variados, como densidade das camadas, tamanho do *kerneis*. Porem uma limitação desse algoritmo é apenas permitir as entradas da primeira população como dados discrepantes, logo, durante as gerações, por exemplo, ao invés da quantidade de filtros da camada de convolução ser, 8, 16, 32 ou 64, o algoritmo gera um valor entre 8 e 64 para os próximos filhos. O algoritmo executa quantas gerações forem passadas, optamos por gerar 5 gerações. O seu retorno é um *array* contento os melhores filhos da ultima geração. Nesse caso, um *array* com 11 saídas, ao invés de 13, como apresentados na Tabela 1, pois a modificação do tamanho dos *kerneis* das camadas de *poolings* estavam gerando erros constantemente. Sobre o algoritmo autoral, as entradas são parecidas, mas as funções são feitas manualmente. Sobre a saida, é similar a anterior, porem foi possivel otimizar o tamanho dos *kerneis* das camadas de *poolings*.

Camada	Hiperparâmetro	1	2	3	4
Camada 2 (convolução)	Número de filtros	64	64	219	81
	Tamanho dos *kernels*	(7, 7)	(7, 7)	(7, 7)	(7, 7)
	Função de ativação	GELU	GELU	GELU	ReLU
Camada 3 (*pooling*)	Tipo de *pooling*	max	max	max	max
	Tamanho do kernel	(2, 2)	(2, 2)	(2, 2)	(2, 2)
Camada 4 (convolução)	Número de filtros	64	64	35	64
	Tamanho dos *kernels*	(5, 5)	(5, 5)	(7, 7)	(7, 7)
	Função de ativação	GELU	GELU	GELU	Softmax
Camada 5 (*pooling*)	Tipo de *pooling*	max	max	avg	max
	Tamanho do kernel	(3, 3)	(3, 3)	(2, 2)	(2, 2)
Camada 7 (densa)	Número de neurônios	128	128	50	256
	Função de ativação	GELU	GELU	Softmax	ReLU
Camada 8 (saída)	Função de ativação	Sigmoid	Sigmoid	Softmax	Sigmoid

Tabela 2. Tabela mostrando os valores resultantes do algoritmo genético. 1 é o AG autoral para o MNIST, 2 é o AG autoral para o *Fashion* MNIST, 3 é o eaSimples para o MINST e 4 é eaSimples para o *Fashion* MNIST

Foi possível perceber que o algoritmo autoral chegou a uma combinação de hiperparametros igual para as duas bases, enquanto o *eaSimples* chegou a combinações diferentes nos dois casos. É interessante notar que para a camada 3, todos os casos foram iguais. Essa consistência dos hiperparâmetros para a camada sugere que esses hiperparâmetros podem ser suficientemente estáveis para bom desempenho do modelo.

Já os resultados finais foram divididos em 4 partes. Duas com o algoritmo *eaSimples* e duas com o algoritmo de implementação própria, cada uma em uma das duas bases de dados.

5.2. Conjunto MNIST

Os resultados obtidos com o algoritmo *eaSimples* para o conjunto de dados MNIST podem ser observados na Figura 1. Foi possível perceber uma queda suave na perda ao passar das épocas, o que demonstra uma aprendizado constante. Já sobre a Figura 2, ela apresenta a matriz de confusão, e foi possível perceber que o as imagens de numero 9 foram todas classificadas como numero 4, porem o 4 não foi classificado como 9.

Figura 1. Perda durante as épocas para o MNIST com o algoritmo *eaSimples*. Imagem elaborada pelos autores.

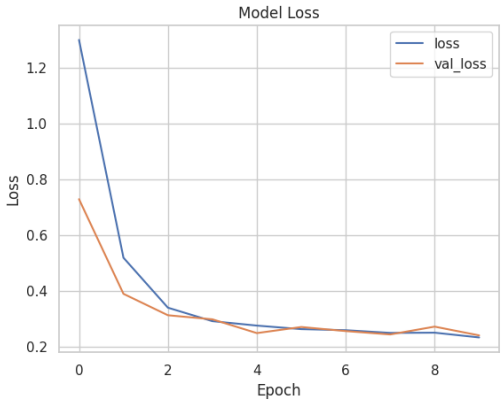
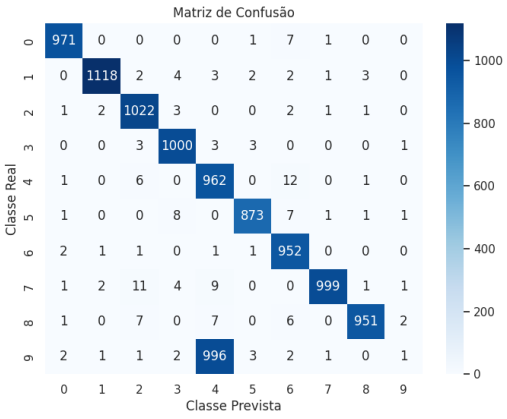


Figura 2. Matriz de confusão para o MNIST com o algoritmo *eaSimples*. Imagem elaborada pelos autores.



Para os resultados obtidos com o algoritmo de implementação própria, pode-se observar na Figura 3 que a perda acontece suavemente, o que indica um aprendizado suave também. Sobre a Figura 4, foi possível observar na matriz de confusão que os dados foram muito bem classificados, tendo menos casos de números 6 classificados erroneamente, em relação a Figura 2.

Figura 3. Perda durante as épocas para o MNIST com o algoritmo autoral. Imagem elaborada pelos autores.

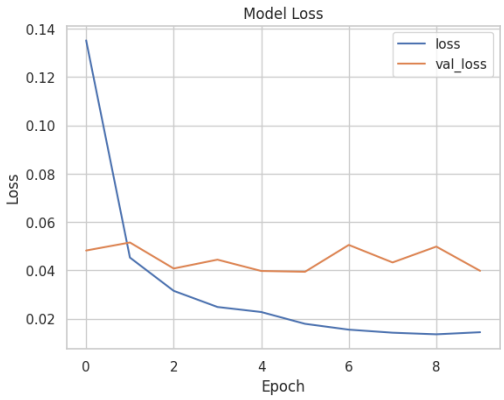
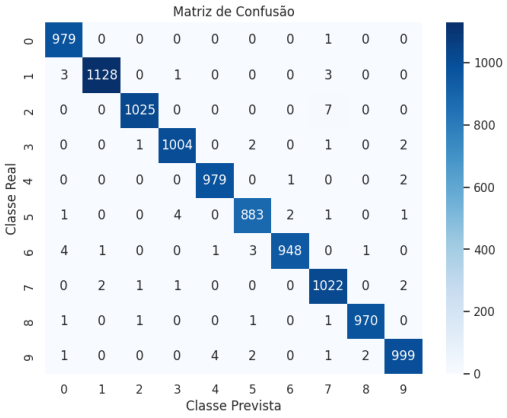


Figura 4. Matriz de confusão para o MNIST com o algoritmo autoral. Imagem elaborada pelos autores.



5.3. Conjunto Fashion MNIST

Os resultados para o algoritmo *eaSimples* com a base de dados *Fashion* MNIST pode ser observado abaixo. Na Figura 5 é possível notar que tanto no teste quanto no aprendizado, os valores foram caindo suavemente durante as épocas, similar a Figura 1. Já na Figura 6, podemos notar sua similaridade a Figura 2, onde os resultados foram muito bons, com alguns pequenos erros na classificação da classe 6.

Figura 5. Perda durante as épocas para o *Fashion* MNIST com o algoritmo eaSimples. Imagem elaborada pelos autores.

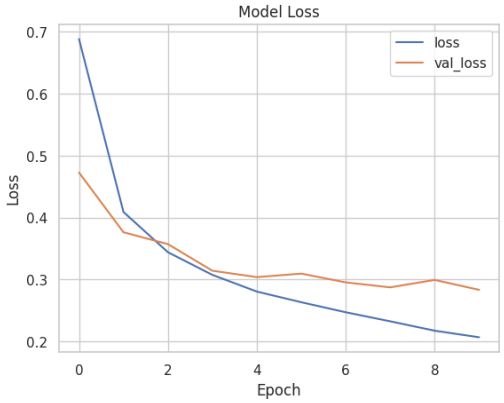
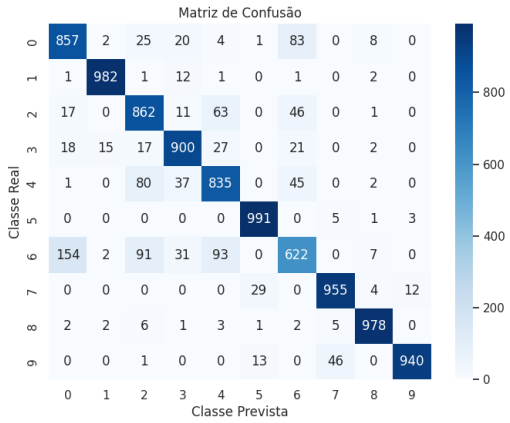


Figura 6. Matriz de confusão para o *Fashion* MNIST com o algoritmo eaSimples. Imagem elaborada pelos autores.



Para os resultados obtidos com o algoritmo de implementação própria, pode-se observar na Figura 7 que a perda acontece suavemente, o que indica um aprendizado suave também. Sobre a Figura 8, foi possível observar na matriz de confusão que os dados foram muito bem classificados, tendo menos casos de números 6 classificados erroneamente, em relação a Figura 2.

Figura 7. Perda durante as épocas para o *Fashion* MNIST para o algoritmo autoral. Imagem elaborada pelos autores.

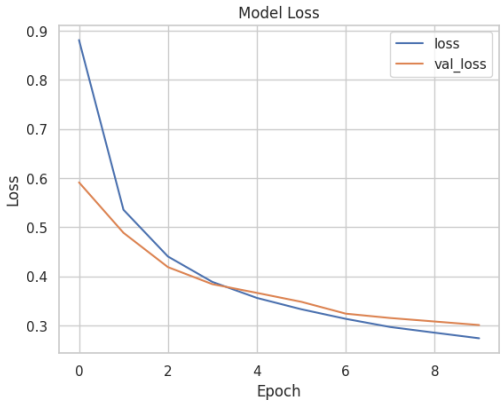
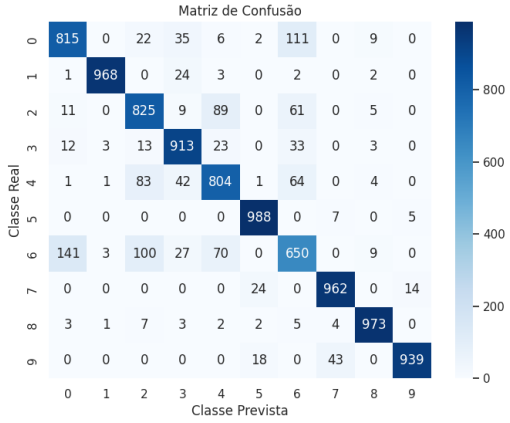


Figura 8. Matriz de confusão para o *Fashion* MNIST para o algoritmo autoral. Imagem elaborada pelos autores.



6. Conclusão

Ambos os métodos utilizados, o algoritmo genético autoral e o eaSimples, demonstraram eficácia na geração de CNNs para o problema em questão, apesar da classificação errônea no numero 9 para o primeiro caso, os outros casos foram bem classificados. Surpreendentemente, nenhum dos métodos apresentou sinais de *overfitting*, sugerindo duas possibilidades intrigantes. Primeiramente, é possível que o problema original não seja tão complexo a ponto de exigir uma otimização extensiva de hiperparâmetros para alcançar um desempenho satisfatório. Alternativamente, poderia ser uma coincidência afortunada que as arquiteturas e os valores de hiperparâmetros escolhidos resultassem em modelos robustos. Essa descoberta destaca a importância da exploração de diferentes abordagens de otimização de hiperparâmetros e sugere uma possível simplificação do processo para problemas similares no futuro. Para trabalhos futuros, é interessante realizar análises mais aprofundadas para compreender melhor o comportamento desses métodos em diferentes contextos e garantir resultados consistentes e confiáveis.]

O código se encontra disponível no repositório *tp_redes_neurais* (https://github.com/rafaugusto20/tp_redes_neurais).

Referências

[Aszemi and Dominic 2019] Aszemi, N. M. and Dominic, P. D. D. (2019). Hyperparameter optimization in convolutional neural network using genetic algorithms. *International Journal of Advanced Computer Science and Applications*.

[Barbosa 2018] Barbosa, F. R. M. (2018). Otimização de hiperparâmetros em algoritmos de arvore de decisão utilizando computação evolutiva.

[Buarque et al. 2022] Buarque, T. M. T., Marinho, M. B. L., and Bernardino Junior, F. M. (2022). Genetic algorithm and pso applied to the choice of hyperparameters of an mlp neural network for non-functional requirements classification. *Research, Society and Development*.

[Faceli et al. 2021] Faceli, K., Lorena, A. C., Gama, J., Almeida, T. A. d., and Carvalho, A. C. P. L. F. d. (2021). *Inteligência Artificial - Uma Abordagem de Aprendizado de Máquina*. LTC.

[Fernandes et al. 2022] Fernandes, S. E. R., da Silva Marques, R. C., de Almeida, J. D. S., de Paiva, A. C., and Junior, G. B. (2022). Otimização de hiperparâmetros de redes neurais profundas para detecção de cardiomegalia em radiografias do tórax. In *Anais do XXII Simpósio Brasileiro de Computação Aplicada à Saúde*, pages 222–233. SBC.

[Jafari-Marandi and Smith 2017] Jafari-Marandi, R. and Smith, B. K. (2017). Fluid genetic algorithm (fga). *Journal of Computational Design and Engineering*, 4(2):158–167.

[LeCun et al. 2010] LeCun, Y., Cortes, C., and Burges, C. (2010). Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann.lecun.com/exdb/mnist*, 2.

[Xiao et al. 2017] Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *CoRR*, abs/1708.07747.

[Yu and Zhu 2020] Yu, T. and Zhu, H. (2020). Hyper-parameter optimization: A review of algorithms and applications.